2015 IEEE 21st International Conference on Parallel and Distributed Systems

FRAME: Fair Resource Allocation in Multi-process Environments

Amin Mohtasham, Ricardo Filipe and João Barreto
INESC-ID Lisboa/ Instituto Superior Técnico, Universidade de Lisboa, Portugal
Email: {amohtasham, rfilipe, jpbarreto}@gsd.inesc-id.pt

***Abstract*—As the technology trend moves toward manufacturing many-core systems with hundreds of processing cores, the problem of efficiently managing multiple parallel jobs on such massively parallel systems becomes increasingly important. With traditional time-sharing each process assumes it is the only running process. This assumption can easily lead to system oversubscription and thereby, losing overall performance due to frequent context switches.**

Space-sharing techniques allocate a certain number of hardware cores to each process and by that *malleable* processes can set their parallelism level to their allocated number of cores, hence avoiding oversubscription. However, finding the optimal spatial allocation is not a trivial task.

In this paper we propose FRAME, a resource allocation technique to maximize a system's overall utility and fairness, running multiple malleable processes with CPU-bound workloads. First, we formalize the resource allocation problem as an NP-hard problem. Then, we use approximation techniques and convex optimization theory to find the optimal solution to the formulated problem, in pseudo-polynomial time.

Our evaluation results show that our method is very fast and efficient in finding the optimal solution to the resource allocation problem. Also, the results suggests that the found solution increases the system's overall utility by 48%, in average, with regard to the best alternative allocation policy.

***Keywords*-Parallel programming, oversubscription, resource management, space-sharing, convex optimization**

I. Introduction

The processor industry is converging to an important technology shift: to go for more, yet simpler, cores per chip. Multi-core architectures are already the norm for most of the commodity computing devices. But, in a near future, many-core chips with hundreds of cores are expected to be an affordable reality [1].

Effectively harnessing such increasingly-parallel machines calls for parallel workloads. As recent studies have observed, there are trends that suggest modern multi/many-core machines will most likely run multiple parallel processes together, rather than a single parallel process alone at a time [2], [3], [4].

The classical approach to collocate multiple processes is through traditional thread schedulers, which time-share the available hardware cores amongst the threads that each process spawns [5]. Time-sharing is transparent to the running processes and allows process to assume it is the only running process in the system. This can easily lead to system *oversubscription*, when the number of computational intensive threads exceeds the number of hardware cores. In an oversubscribed system, time-sharing incurs non-negligible costs. These costs are due to frequent context switches between different threads and more cache-trashing. Additionally, the system's throughput can be highly damaged when, for example, a running thread that holds a lock is preempted while the other running threads cannot move forward because they are waiting for that lock to be released.

More recently, new solutions have proposed a distinct approach to the problem: *space-sharing* together with *time-sharing* [6]. In this 2-level approach, first, the hardware cores are spatially divided between the processes. Next, the cores allocated to each process are time-multiplexed between the threads inside that process [6], [7].

This strategy fits particularly well with *malleable* applications with CPU-bound workloads. Such applications are flexible and can set their parallelism level at the beginning or throughout their execution [8]. Knowing the number of allocated cores, a malleable application simply adjusts its parallelism level to its hardware share and amortizes the aforementioned negative effects of time-sharing.

Hence, it is desirable to solve the following problem: to fairly allocate hardware cores to processes in such a way that system overall utility is maximized. Intuitively, an allocation is fair when no process starves and, more importantly, resources that a process receives correspond to its demand. Also, a system's overall utility is defined as a function of each process's utility in that system.

In order to find a fair allocation that meets the applications' processing demands, the applications must provide the scheduler with the information about their workloads [7]. Such information can be gathered through an online or offline profiling phase. However, given such information, finding the optimal core allocation is not trivial for two main reasons: (i) it requires solving an NP-hard optimization problem, and (ii) quantifying and ensuring fairness is not trivial.

To have more insights about the time complexity of finding the optimal core allocation for co-exiting processes, assume a 64-core system running multiple parallel processes with known workloads. Depending on the resource allocation, each process exhibits a throughput which can be translated into that process's utility. The goal of resource allocation is dividing the 64 cores between the processes in a way that the system's overall utility is maximized. Figure 1 shows the time needed by a brute-force search to explore all the feasible domain of this following example. The values are in logarithmic scale. Assuming each iteration

1521-9097/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPADS.2015.81

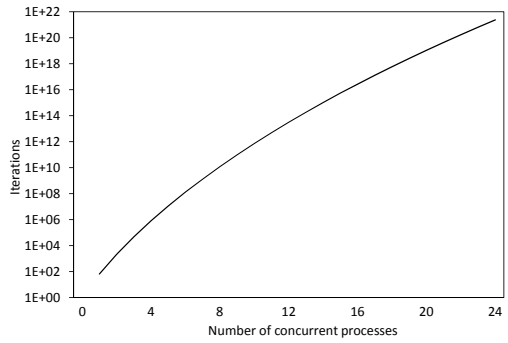


Figure 1: The cost of exhaustive search to find the optimal resource allocation.

takes only 1 nanosecond (in practice it is more than couple of microseconds), an exhaustive search to find the optimal core allocation for 24 concurrent processes takes up to 75000 years. In fact, the number of iterations is equal to $\mathbf{C}_{\mathcal{P}}^{(\mathcal{P}+\mathcal{C})}$, where $\mathcal{P}$ and $\mathcal{C}$ denote the number of concurrent processes and hardware cores, respectively, and $\mathbf{C}$ is the combination function.

In this paper, we make the following contributions:

- We use the utility and the fairness notions of *Nash bargaining solution* (NBS) [9] to propose a first formal definition of the core-allocation problem for multi-process parallel systems that takes both utility and fairness into account.
- By relying on convex optimization theory [10], we propose FRAME, an efficient and fast method to find the solution to the formalized problem, in pseudo-polynomial time, for malleable parallel applications whose scalability curves are concave.
- By experimental evaluation we show that FRAME leads the system to a configuration that yields both maximized utility and fairness an increases the system's overall utility by 48%, on average, with regard to the best alternative allocation policy.

The remaining of the paper is structured as follows. Section II surveys the related work. Section III then introduces the formal notions that allow us to cleanly define the problem of core allocation. Section IV transforms the formulated problem into an approximated problem that makes it possible for us to use convex optimization theory. Section V solves the approximated problem and proposes an iterative algorithm to find the optimal solution. Section VI then evaluates the proposed solution, comparing it to currently well-known techniques. Finally, Section VII draws conclusions and discusses future work.

II. Related Work

The problem of finding the optimal core allocation has been studied by the research community in the last couple of years.

PACORA [11] is a general resource allocation framework for manycore machines. This framework can be used to allocate different types of resources to the processes. In PACORA, each process defines a penalty function and then the resource allocator minimizes the total aggregated penalty in the system. The minimization is done using the convex optimization theory. However, apart from being too generic, the aggregation approach taken by PACORA may lead to an unfair allocation where some processes starve.

Creech *et. al.* [4] propose SCAF. SCAF is a core allocation framework that was proposed to efficiently allocate cores to malleable parallel processes. The core allocation in SCAF is done by maximizing the system's overall utility. SCAF prevents starvation by making sure that each process receives at least one processing core. However, this cannot be interpreted as fairness and SCAF's problem formulation imposes a very weak notion of fairness. For example, in a system with hundreds of cores, allocating just a single core to a multi-threaded process could be highly unfair. Furthermore, SCAF assumes a simple scalability model, in which all processes are scalable up to the number of available cores. In many real applications, parallel workloads may stop scaling at a certain level, due to synchronization overhead and contention over shared resources [12].

Kazi *et. al.* [13] propose a core allocation technique for co-existing processes with parallel loops. Their method is limited to completely scalable loops without loop-carried dependencies. Also, this method does not assure fair allocation of cores to co-existing processes.

Harris *et. al.* [3] introduce Callisto, a resource management level for parallel runtimes. Callisto prevents processes from interfering with each other in terms of claimed hardware resources. Paralell applications announce their hardware demands and divide the resources collaboratively. One of the root blocks of Callisto is a spatial allocation policy, which the authors leave as an open option. FRAME can be used as the spatial allocation algorithm in Callisto, thereby leveraging Callisto with optimal and fair spatial allocation.

Didona *et. al.* [14] have proposed an adaptive method to dynamically find the best parallelism point for shared-memory transactional memory and distributed transactional memory. They use transaction commit-rate as the performance measure and try to maximize the throughput by dynamically changing the number of active threads, taking a hill-climbing approach. In Section VI, we compare our results to those of this method as a state-of-the art dynamic mechanism to find the optimum parallelism level.

III. The Fair Resource Allocation Problem

To discuss fairness, first, we need to define the fairness criterion. Our definition of a fair system is based on what was proposed in Nash bargaining solution (NBS) [9].

In a system with $\mathcal{P}$ running processes, each running workload w_p, we define the utility of process p ($\mathcal{U}_p$), as the ratio between the process' current throughput ($T_p(w_p)$), and the maximum achievable throughput for the workload w_p,

in a single process configuration ($T^*(w_p)$). In other words, $\mathcal{U}_p(w_p) = \frac{T_p(w_p)}{T^*(w_p)}$. $\mathcal{U}_p$ is always positive and is equal or less than 1.

Based on NBS, a fair resource allocation maximizes the system's overall utility. This utility is defined by the product of all process' utility value (i.e. $\prod_{p=1}^{\mathcal{P}} \mathcal{U}_p(w_p)$). For example, it can be easily shown that in a contended system running identical processes, equally sharing the hardware between the processes maximizes the system's overall utility.

A. Resource Allocation Model

The resource allocation environment is modeled as follows. We consider a parallel system with $\mathcal{C}$ hardware cores, where $\mathcal{P}$ collocated processes run.

We assume that the workloads are generally CPU-bound and so each software thread is able to completely use a single hardware core. Hence, in a multi-process environment, computational power (i.e. hardware cores) is the main resource that is to be allocated to co-existing processes.

Each process p_i wishes to execute a given workload, w_i, with $\mathcal{M}_i$ parallel threads. $\mathcal{M}_i$ is the number of parallel threads by which p_i reaches its maximum throughput running workload w_i in a single-process environment (optimal parallelism level). However, since resources are limited, the system might not provide p_i with the desired number of processing cores. More precisely, we consider that, at each moment, process p_i is allowed to run τ_i active threads, where $0 \leq \tau_i \leq \mathcal{M}_i$.

An *allocation* shows the number of threads in all processes in vector form by $\vec{\tau} = (\tau_i, i = 1, ..., \mathcal{P})$. Similarly, the vector $\vec{\mathcal{M}} = (\mathcal{M}_i, i = 1, ..., \mathcal{P})$ shows the optimal parallelism levels for the process' workload in the single-process mode.

The utility function of each process p_i is represented as a function of the number of threads allocated to that process $\mathcal{U}_i(\tau_i)$, where $\tau_i \in [0, \mathcal{M}_i]$, $\mathcal{U}_i \in [0, 1]$.

We consider workloads that scale sub-linearly until their optimal parallelism level. However, sub-linear scalability is more common due to the increases in contention between parallel threads or thread management overhead and many real-life workloads fall into this category [12], [15]. Therefore, for a given workload w_i, its utility function $\mathcal{U}_i(\tau_i)$ is strictly increasing, when $\tau_i \in [0, \mathcal{M}_i]$.

We assume that there is a central entity responsible for allocating cores to the parallel processes. This entity knows about the number of processes and each process' workload and makes decisions based on this information.

The goal of resource allocation is to ensure that a set of parallel processes leaves exactly one software thread pinned to each core, while the overall utility of the system is maximized. In other words, the resource allocator efficiently allocates $\mathcal{C}$ hardware cores to $\mathcal{P}$ parallel process, where the total number of software threads does not exceed $\mathcal{C}$. Also, the resource allocator makes sure that no process receives more cores than its corresponding $\mathcal{M}_i$. These constraints can be expressed as:

$$\sum_{i=1}^{\mathcal{P}} \tau_i \leq \mathcal{C}, \tag{1}$$

$$\tau_i \leq \mathcal{M}_i, i = 1, ..., \mathcal{P}. \tag{2}$$

For the sake of notation, we rewrite constraints (1) and (2) in vector form as:

$$\vec{1}^{\intercal}\vec{\tau} \leq \mathcal{C}, \tag{3}$$

$$\vec{\tau} \leq \vec{\mathcal{M}}, \tag{4}$$

where $^{\intercal}$ denotes a transposed vector/matrix.

B. Problem Formulation

The goal of the resource allocation is to maximize the system's overall utility. Since the maximization should be done over the feasible region, which is characterized by (4) and (3), the optimization problem can be formulated as:

$$\begin{aligned} \underset{\vec{\tau}}{\text{maximize}} \quad & \prod_{i=1}^{\mathcal{P}} \mathcal{U}_i(\tau_i) \\ \text{subject to} \quad & \vec{1}^{\intercal}\vec{\tau} \leq \mathcal{C} \text{ and } \vec{\tau} \leq \vec{\mathcal{M}}, \end{aligned} \tag{5}$$

where $\mathcal{U}_i$ denotes the ith process's utility function.

Considering each $\mathcal{U}_i(\tau_i)$ being strictly increasing on $[0, \mathcal{M}_i]$, we use a logarithmic monotonic transform and, without loss of correctness, the problem (5) is rewritten as:

$$\begin{aligned} \underset{\vec{\tau}}{\text{maximize}} \quad & \sum_{i=1}^{\mathcal{P}} \ln \mathcal{U}_i(\tau_i) \\ \text{subject to} \quad & \vec{1}^{\intercal}\vec{\tau} \leq \mathcal{C} \text{ and } \vec{\tau} \leq \vec{\mathcal{M}}. \end{aligned} \tag{6}$$

This problem is an integer linear programming (ILP) problem, which is NP-hard and can not be solved in polynomial time, in its general case.

In order to find the solution to (6), we use convex optimization theory [10].

IV. Approximation to the Problem

In this section we describe how we transform the above formulated problem into an approximated problem that makes it possible for us to use convex optimization theory.

A. Approximation Function

To efficiently solve Problem (6), first, we need to approximate $\mathcal{U}_i(\tau)$ with a well-behaving function, $\mathcal{H}_i(\tau)$.

As stated earlier, a workload w_i scales sub-linearly and its utility function, $\mathcal{U}_i$, is a strictly increasing function and ranges from 0 to 1.

The above features suggest that an ideal approximation function $\mathcal{H}$ must be strictly increasing and concave (i.e. twice-continuously differentiable).

It is easy to prove that with such an approximation function, Problem (6) admits a unique maximizer [10].

Using the aforementioned features, a good candidate is the cumulative distributed function (CDF) of exponential distribution, which is defined as: $1 - e^{-\lambda x}$.

In this paper, we use the generic form of the exponential distribution's CDF to define $\mathcal{H}$:

$$\mathcal{H}(x, \alpha, \beta, \gamma) = \gamma - e^{-\alpha(x-\beta)}, \tag{7}$$

where α, β and γ are approximation parameters.

B. Deriving Approximation Parameters

In order to approximate $\mathcal{U}_i$ with $\mathcal{H}_i$, approximation parameters should be selected so as to minimize the error between $\mathcal{U}_i$ and $\mathcal{H}_i$. Toward this end, several error metrics can be employed. We used *mean square error* (MSE), which is a well-known error metric [16]. We define the MSE as the square difference between the original function (i.e. $\mathcal{U}_i$) and its approximated function (i.e. H_i):

$$\mathsf{MSE}(\mathcal{U}, \mathcal{H}) = \int_{\mathcal{D}} \big((\mathcal{U}(x) - \mathcal{H}(x, \alpha, \beta, \gamma)\big)^2 dx, \tag{8}$$

where $\mathcal{D}$ is the domain of interest and for $\mathcal{U}_i$, $\mathcal{D}_i$ is equal to $[0, \mathcal{M}_i]$ and $\mathcal{D} \subset \mathbb{Z}$ ($\mathbb{Z}$ is the set of all integers).

Based on $\mathsf{MSE}(\mathcal{U}, \mathcal{H})$ defined above, the approximation parameters α, β and γ can be found as the solution to the following optimization problem:

$$\underset{\alpha,\beta,\gamma}{\text{minimize}} \quad \mathsf{MSE}(\mathcal{U}, \mathcal{H}). \tag{9}$$

Even if there exists a closed form universal solution to (9), it is quite cumbersome and challenging. Furthermore, $\mathsf{MSE}(\mathcal{U}, \mathcal{H})$ might not be a convex function for all $\mathcal{U}_i$ functions over all desired $\mathcal{D}_i$ domains. Therefore, we consider Problem (9) as a non-convex problem, which can not be solved by convex optimization theory in general case.

Since Problem (9) is unconstrained, we can apply iterative methods to solve it. Amongst iterative methods, the gradient descent method and its variants are simple and computationally efficient. However, as Problem (9) is a non-convex problem, there is no guarantee to find the global optima. Instead, in order to solve Problem (9), we resort to using randomized methods, such as particle swarm optimization (PSO) [17].

C. The Approximated Problem

With the above approximation and finding the approximation parameters, α_i, β_i and γ_i, for each process' utility function, $\mathcal{U}_i$, Problem (6) can be rewritten as:

$$\begin{aligned} \underset{\vec{\tau}}{\text{maximize}} \quad & \sum_{i=1}^{\mathcal{P}} \ln\left(\gamma_i - e^{-\alpha_i(\tau_i - \beta_i)}\right) \\ \text{subject to} \quad & \vec{1}^{\intercal}\vec{\tau} \leq \mathcal{C} \text{ and } \vec{\tau} \leq \vec{\mathcal{M}}. \end{aligned} \tag{10}$$

In the following section we find the solution of Problem (10), using convex optimization theory.

V. Finding the Optimal Solution

In this section, we solve Problem (10) using convex optimization theory and dual-based approaches. To do so, first we need to prove that Problem (10) is a convex problem in its domain.

A. Concavity

In order for Problem (10) to admit a unique maximizer and to find that maximizer by convex optimization theory, each $\ln \mathcal{H}_i(\tau)$ should be a concave function throughout its domain. Since $\ln$ is a strictly increasing and concave function, the $\ln$ transformation preserves the concavity. Therefore, we only need to proof the concavity of the $\mathcal{H}_i$ functions.

For $\mathcal{H}_i(\tau)$ to be concave, we need to prove that $\mathcal{H}_i(\tau)$ is twice-continuously differentiable and its second derivative must be negative within its domain, which is shown in the following:

$$\begin{aligned} \frac{d^2\mathcal{H}_i}{d\tau^2} &= \frac{d^2(\gamma_i - e^{-\alpha_i(\tau-\beta_i)})}{d\tau^2} \\ &= -\alpha_i^2 e^{-\alpha_i(\tau-\beta_i)} < 0; \quad \alpha_i \neq 0. \end{aligned} \tag{11}$$

Since $\mathcal{H}_i(\tau)$ is strictly concave, we deduce that $\mathcal{H}(\vec{\tau})$, which is the non-negative weighted sum of strictly concave functions, is strictly concave.

Also, the constraints in Problem 10 include affine functions, hence they are convex.

By this, we deduce Problem 10 is convex problem. Since the feasible set is compact and the objective is strictly concave, there exists a unique maximizer for Problem 10.

B. Optimality Conditions

To derive optimality conditions, we write the Lagrangian of Problem (10) as follows [10]:

$$L(\vec{\tau}, \vec{\mu}, \nu) = \sum_{i=1}^{\mathcal{P}} \ln \mathcal{H}_i(\tau_i) - \vec{\mu}^{\intercal}(\vec{\tau} - \vec{\mathcal{M}}) - \nu(\vec{1}^{\intercal}\vec{\tau} - \mathcal{C}), \tag{12}$$

where $\vec{\mu} = (\mu_i, i = 1, ..., \mathcal{P})$ is the vector of positive Lagrange multipliers associated with Constraint (4) and, similarly, ν is a positive Lagrange multiplier associated with Constraint (3).

The answer for a convex optimization problem must satisfy the Karush-Kuhn-Tucker (KKT) conditions [10]. The KKT conditions for optimal primal variable $\vec{\tau}^*$ and dual variables, $\vec{\mu}^*$ and ν^*, are:

$$\nabla_{\vec{\tau}} L(\vec{\tau}^*, \vec{\mu}^*, \nu^*) = \vec{0}, \tag{13}$$

$$\vec{\mu} \geq \vec{0} \text{ and } \nu \geq 0, \tag{14}$$

$$\vec{\mu}^{*\intercal}(\vec{\tau}^* - \vec{\mathcal{M}}) = 0 \text{ and } \nu^*(\vec{1}^{\intercal}\vec{\tau^*} - \mathcal{C}) = 0, \tag{15}$$

where $\vec{0}$ is a vector with all zero. Condition (13) ensures there is no feasible directions that improves the objective function. Conditions in (14) depict dual feasibility, which

say the dual variables (*a.k.a.* shadow price) are non-negative values. Finally, conditions in (15) tell us if a dual variable is positive, then the associated primal constraint must be binding and if a primal constraint fails to bind, then the associated dual variable must be zero.

Solving (13) to find the stationary points of the Lagrangian yields:

$$\frac{\partial L}{\partial \tau_i} = \frac{\alpha_i e^{-\alpha_i(\tau_i-\beta_i)}}{\gamma_i - e^{-\alpha_i(\tau_i-\beta_i)}} - \mu_i - \nu = 0. \tag{16}$$

After solving (16), for optimal primal variable τ_i^*, we have:

$$\tau_i^* = \beta_i - \frac{1}{\alpha_i} \ln\left(\frac{\gamma_i(\mu_i+\nu)}{\alpha_i+\mu_i+\nu}\right). \tag{17}$$

In order to solve Problem (10) using its dual, we have to obtain the Lagrange dual function. The dual function $D(\vec{\mu},\nu)$ is defined as the maximum of Lagrangian $L(\vec{\tau},\vec{\mu},\nu)$ over the primal variable $\vec{\tau}$, for given $\vec{\mu}$ and ν. Thus, $D(\vec{\mu},\nu)$ can be expressed as:

$$D(\vec{\mu},\nu) = \max_{\vec{\tau}} L(\vec{\tau},\vec{\mu},\nu). \tag{18}$$

Based on KKT conditions and using (17), for $D(\vec{\mu},\nu)$ we have:

$$D(\vec{\mu},\nu) = L(\vec{\tau}^*,\vec{\mu},\nu). \tag{19}$$

Then, the dual problem is formulated as [10]:

$$\underset{\vec{\mu}\geq\vec{0},\ \nu\geq 0}{\text{minimize}} \quad D(\vec{\mu},\nu) \tag{20}$$

In the next section, we solve the above dual problem, using iterative methods.

C. Solving the Dual

In this section we solve the dual problem (20) using gradient projection algorithm. Gradient projection algorithm is an iterative method that adjusts $\vec{\mu}$ and ν in the direction to the gradients of dual function, i.e. $\nabla D(\vec{\mu},\nu)$. To be more specific, in the kth iteration, the iterative algorithm updates the dual variables $\mu_i^{(k)}$ and $\nu^{(k)}$ as follows:

$$\mu_i^{(k+1)} = \left[\mu_i^{(k)} - \psi\frac{\partial D(\vec{\mu}^{(k)},\nu^{(k)})}{\partial \mu_i}\right]^+ ; i = 1,...,\mathcal{P} \tag{21}$$

and

$$\nu^{(k+1)} = \left[\nu^{(k)} - \psi\frac{\partial D(\vec{\mu}^{(k)},\nu^{(k)})}{\partial \nu}\right]^+, \tag{22}$$

where $[z]^+ = \max\{z,0\}$ and $\psi > 0$ is a sufficiently small constant step-size.

Using Danskin's theorem [18], partial derivatives of $D(\vec{\mu},\nu)$ are given by:

$$\frac{\partial D(\vec{\mu},\nu)}{\partial \mu_i} = \mathcal{M}_i - \tau_i \tag{23}$$

and

$$\frac{\partial D(\vec{\mu},\nu)}{\partial \nu} = \mathcal{C} - \vec{1}^\intercal\vec{\tau} = \mathcal{C} - \sum_{i=1}^{\mathcal{P}} \tau_i. \tag{24}$$

Substituting (23) and (22) in (21) and (22, respectively, yields:

$$\mu_i^{(k+1)} = \left[\mu_i^{(k)} - \psi\big(\mathcal{M}_i - \tau_i^{(k)}\big)\right]^+ \tag{25}$$

and

$$\nu^{(k+1)} = \left[\nu^{(k)} - \psi\left(\mathcal{C} - \sum_{i=1}^{\mathcal{P}} \tau_i^{(k)}\right)\right]^+, \tag{26}$$

where $\tau_i^{(k)}$ is obtained using (17), given $\mu_i^{(k)}$ and $\nu^{(k)}$.

Algorithm 1 The iterative algorithm

Initialization:
Calculate $(\alpha_i,\beta_i,\gamma_i)$;

Main Loop:
repeat
  $\nu^{(k+1)} = \left[\nu^{(k)} - \psi\left(\mathcal{C} - \sum_{i=1}^{\mathcal{P}} \tau_i^{(k)}\right)\right]^+$
  for $i = 1$ **to** $\mathcal{P}$ **do**
    $\mu_i^{(k+1)} = \left[\mu_i^{(k)} - \psi\big(\mathcal{M}_i - \tau_i^{(k)}\big)\right]^+$
    $\tau_i^{(k+1)} = \beta_i - \frac{1}{\alpha_i}\ln\left(\frac{\gamma_i(\mu_i^{(k+1)}+\nu^{(k+1)})}{\alpha_i+\mu_i^{(k+1)}+\nu^{(k+1)}}\right)$
  end for
until $\max\left|\tau_i^{(k+1)} - \tau_i^{(k)}\right| < \epsilon$

Output:
Let every process knows about its thread count.

D. The Iterative Algorithm

In this section we propose an algorithm based on the iterative solution obtained in the previous section. From the proposed solution, equations (17), (21) and (22) are jointly used in the proposed algorithm. Equation (17) calculates the optimal number of threads in each process and the later two are used to update Lagrange multipliers.

Algorithm 1 shows how the iteration algorithm works. At the initialization step, the approximation parameters (α,β,γ) are calculated for each process, solving Problem (9) described in Section IV.

Then using the approximation parameters, the iterative algorithm proceeds as follows. At iteration step k, the Lagrange multipliers are updates using equations (21) and (22). Then, the optimal number of threads for each process is updated, based on the new Lagrange multipliers value, using (17). The algorithm keeps repeating this procedure until it converges to the optimal point.

The final allocation comprises integer values. Hence, $\vec{\tau}^*$ is rounded down, and the remaining hardware cores are divided between the processes that were rounded down the most.

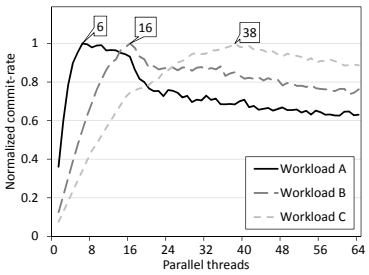


Figure 2: The scalability/utility graph of the experimental workloads. Each workload is marked with its optimal parallelism level.

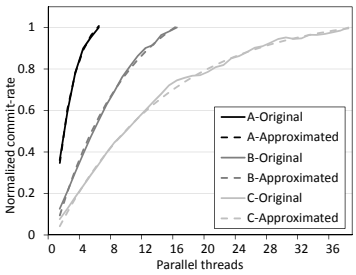


Figure 3: The curve-fitting's accuracy using PSO. All the approximated functions very well fit the corresponding curves.

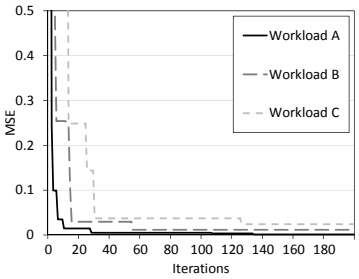


Figure 4: The curve-fitting's convergence speed using PSO to minimize MSE. Each iteration takes almost 250 microseconds.

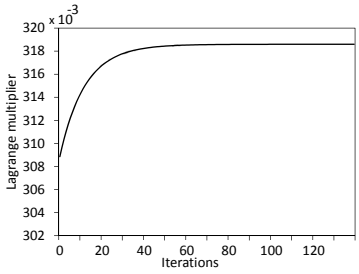


Figure 5: The FRAME's convergence for 24 concurrent processes. Each iteration takes almost 30 microseconds.

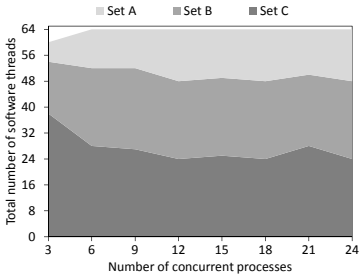


Figure 6: Thread allocation results for 3 up to 24 concurrent processes divided in 3 sets.

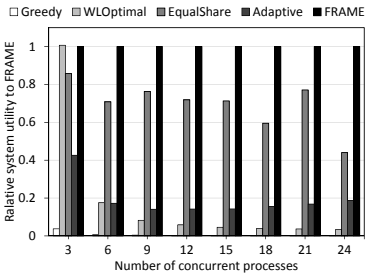
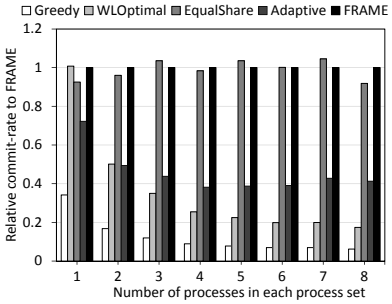
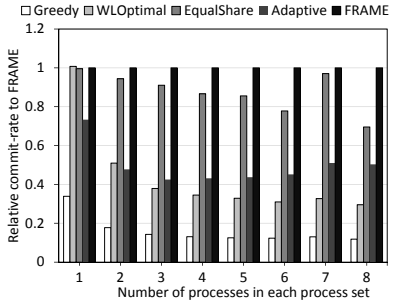


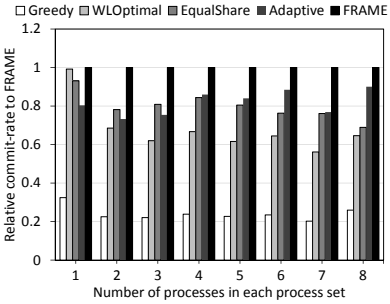
Figure 7: The system's utility with different policies for 3 up to 24 concurrent processes divided in 3 sets.



(a) Workload A



(b) Workload B



(c) Workload C

Figure 8: Relative commit-rate in each process-set.

VI. Experimental Evaluation

Our method is applicable to any set of processes with CPU-bound workloads. The only requirement for the workloads is the concavity of their scalability/utility curve. Although this requirement may narrow down the types of the workloads we can use, there is still a wide range of parallel workloads that fit into this category. This stems from the fact that most of the parallel workloads monotonically scale (sub)linearly up until they reach their maximum throughput.

Nevertheless, there are cases when a workload loses its monotony because of some external factors, such as NUMA effect, when a process starts using a new socket on a NUMA machine. In such cases, the approximated functions and thereby the final solution lose their accuracy.

The workloads used in our evaluation are generated by running a *red-black tree* benchmark on top of the NOrec STM [19], as the underlying parallelism runtime.

In this benchmark, each transaction either looks up, inserts or deletes an element in the red-black tree. By changing the ratio of look-up transactions against those that update the tree, we change the contention level and hence the natural parallelism level.

We consider three different workloads, named A, B and C. Figure 2 shows the scalability graph of these workloads. These graphs are obtained by running the workloads in a host environment consisting of 4x AMD Opteron 6272

CPUs, each with 16 cores, where each core runs one thread, resulting in 64 hardware cores and total memory of 32GB, running linux kernel 3.2.0-58.

In the scalability graphs, we measure the throughput, in terms of transaction commit-rate, as the number of parallel threads grows from 1 to 64. To obtain the utility functions, the values are normalized relatively to the peak throughput that was measured in each workload. Each graph is also marked with its optimal parallelism level, at which the workload exhibits its highest throughput in a single-process environment.

It should be noted the choice of the host machine, the parallelism runtime and the application does not affect our results' correctness. It stems from the fact that our technique is only concerned with the utility function acquired by any arbitrary parallel application with any workload running and it finds the optimal resource allocation for that combination that maximizes the system's overall utility.

A. Workload Approximation Functions

As discussed earlier in Section IV, we use PSO to solve an MSE problem to find the best fitting function for each workload's utility graph.

Our PSO implementation is a simple one with no parallelism and each iteration takes almost 250 microseconds.

Figure 4 shows the convergence of PSO in solving the MSE minimization problem for each workload. Although PSO is an evolutionary method, the observed convergence behavior suggests the fact that PSO performs very fast in order to find the best fitting approximation function. For each workload, PSO almost converges in less than 60 iteration steps which translates into 15 milliseconds.

Furthermore, Figure 3 shows the high accuracy of the approximation functions found by PSO. All the approximation functions highly fit the corresponding original ones.

B. Core Allocation Policies

We compare FRAME's results to those gathered by the following allocation policies:

- **Greedy**: This is the simplest baseline policy which is used by many parallel systems. In this policy, each process tries to take over the whole hardware by defining as many parallel threads as hardware cores.
- **WLOptimal:** Using this policy, each process sets its parallelism level to its workload's optimal level, depicted in Figure 2. It is worth mentioning that, this policy is not the optimal policy and like with the Greedy policy, the total number of software threads can easily go beyond the number of hardware cores.
- **EqualShare:** In this policy, each process takes an equal share of the hardware, regardless of its workload. To be more specific, in a system with $\mathcal{C}$ hardware cores and $\mathcal{P}$ concurrent processes, each process receives $\lfloor \frac{\mathcal{C}}{\mathcal{P}} \rfloor$ cores. This policy ensures that the total number of software threads never exceeds the number of hardware cores.
- **Adaptive:** We use an adaptive parallelism technique, proposed in [14], as a state-of-the-art policy, which is described previously in Section II.

C. Evaluation Configurations

In order to measure the efficiency of different allocation policies, we evaluate them in a heterogeneous environment and with a varying number of parallel processes.

For the sake of heterogeneity, in each experiment, the parallel processes are divided into three sets of processes with the same size. Hence, the total number of coexisting processes is a multiple of three. Each process-set is associated with either of the workloads A, B or C. All processes inside a process-set are homogeneous and they execute the associated workload, but the each process-set runs a distinct workloads.

Throughout different experiments, we vary the number of total processes from 3 to 24, which means 1 to 8 processes inside each process-set.

D. Convergence of the Iterative Method

The convergence of the Lagrange multiplier shows how Algorithm 1 converges to the optimal solution to the core allocation problem.

Figure 5 shows the evolution of the Lagrange multiplier, where the iterative algorithm is set to find the optimal solution. In this experiment, the system runs 24 concurrent processes (i.e. 8 processes in each process-set).

The algorithm converges to the optimal solution after almost 50 iteration steps. This convergence is considerably faster than the equivalent exhaustive search algorithm, which needs to take about 23×10^{20} cases into account to find the optimal solution.

The step-size, ψ, is set to 0.0001.

E. Core Allocation Results

Figure 6 depicts the resource allocation results of FRAME. This figure represents the total number of threads (cores) allocated to each process-set for a single policy. We can see that FRAME has a notion of fairness where neither of the processes starve even in a highly contented configuration with 24 processes. Also, more demanding processes (e.g., Process-set C) receive more resources comparing to the other processes, while the less demanding ones still get their own share of the system and do not starve.

F. Performance Results

In all the performance graphs, the values are divided by those measured by the FRAME policy.

Figure 7 shows the system's total utility employing either of the policies. This figure shows that FRAME and EqualShare, which strictly keep the total number of threads equal to or less than the number of h/w cores, perform much better than the other core allocation policies. Also, from this figure, it is evident that FRAME always reaches the highest utility level. In fact, FRAME increases the overall system's utility

up to 127%, and 43% in average, compared to the second best policy, which is EqualShare.

Also, Figure 8 depicts the effect of employing either of the discussed policies on each process-set's throughput (i.e. commit-rate).

In this figure, for all the workloads, FRAME either exhibits the best throughput or it performs comparable to the best policy.

It is noteworthy that in the 3-process configuration (i.e. 1 in each process-set), WLOptimal leads to the optimal resource allocation, since it does not oversubscribe the system. This justifies the fact that WLOptimal performs as well as FRAME in this configuration.

These performance results prove that in fact FRAME is the best core allocation policy to date,

VII. Conclusion and Future Work

Current trends suggest that modern parallel machines are likely to run multiple parallel processes together. Hence, it becomes increasingly important to devise effective approaches to adjust each process' parallelism in order to maximize the system's overall utility while ensuring fairness between the parallel processes. However, the state-of-the-art solutions are simply tailored to maximize the overall throughput and neglect the fairness requirement of a system.

In this paper, we proposed FRAME as a fair resource allocation method for multi-process environments. We used the notion of utility and fairness from the Nash bargaining problem and we formalized the core allocation problem for computationally intensive co-existing parallel processes. In this problem hardware cores are to be allocated to the parallel processes in a way that the system's overall utility is maximized, while fairness is preserved. Then, we solved the core allocation problem by employing convex optimization theory and we proposed an iterative algorithm that can be used to solve this problem in practice.

By evaluating against the state-of-the-art policies, we showed how FRAME increases the system's overall utility by 48%, in average, with regard to the best alternative allocation policy.

Although our evaluation uses transactional memory as the underlying parallel runtime, FRAME is easily applicable to any parallel programming paradigm. The key insight is that, as long as there are meaningful and precise ways of measuring the utility function of each process, it can serve as the input to our method.

References

[1] S. Borkar, "Thousand core chips: a technology perspective," DAC'07, ACM, 2007.

[2] S. Peter, A. Schüpbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe, "Design principles for end-to-end multicore schedulers," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, USENIX Association, 2010.

[3] T. Harris, M. Maas, and V. J. Marathe, "Callisto: co-scheduling parallel runtime systems," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, ACM, 2014.

[4] T. Creech, A. Kotha, and R. Barua, "Efficient multiprogramming for multicores with scaf," MICRO-46, ACM, 2013.

[5] A. S. Tanenbaum, "Modern operating systems," 2009.

[6] D. Feitelson, "Job scheduling in multiprogrammed parallel systems," tech. rep., IBM Research Report RC 19790 (87657) 2nd Revision, 1997.

[7] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson, "Adaptive scheduling with parallelism feedback," PPoPP'06, ACM, 2006.

[8] D. Feitelson and L. Rudolph, "Toward convergence in job schedulers for parallel supercomputers," in *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1996.

[9] K. Binmore, A. Rubinstein, and A. Wolinsky, "The nash bargaining solution in economic modelling," *The RAND Journal of Economics*, 1986.

[10] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2009.

[11] S. L. Bird and B. Smith, "Pacora: Performance aware convex optimization for resource allocation," in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar: Posters)*, 2011.

[12] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," IISWC'08, 2008.

[13] I. Kazi and D. Lilja, "A comprehensive dynamic processor allocation scheme for multiprogrammed multiprocessor systems," in *Proceedings of the 2000 International Conference on Parallel Processing*, pp. 153–161, IEEE Computer Society, 2000.

[14] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," in *Networked Systems*, Springer Berlin Heidelberg, 2013.

[15] R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: A benchmark for software transactional memory," EuroSys'07, ACM, 2007.

[16] N. R. Draper, H. Smith, and E. Pownell, *Applied regression analysis*, vol. 3. Wiley New York, 1966.

[17] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, IEEE, 1995.

[18] D. P. Bertsekas, "Nonlinear programming," 1999.

[19] L. Dalessandro, M. F. Spear, and M. L. Scott, "Norec: Streamlining stm by abolishing ownership records," PPoPP'10, ACM, 2010.